VISUALIZATION OF SIGNALS AND ALGORITHMS IN KRONOS

Vesa Norilo

Centre for Music and Technology
Sibelius Academy
Helsinki, Finland
vnorilo@siba.fi

ABSTRACT

Kronos is a visual-oriented programming language and a compiler aimed at musical signal processing tasks. Its distinctive feature is the support for functional programming idioms like closures and higher order functions in the context of high performance real time DSP. This paper examines the visual aspect of the system. The programming user interface is discussed, along with a scheme for building custom data visualization algorithms inside the system.

1. INTRODUCTION

Much research effort has been spent to pursue visual programming of audio algorithms. Signal flows are well suited for graphical representation, and graphical user interfaces for programming tend to lower the barrier to entry for non-traditional programmers.

On the other hand, data visualization is key in various audio tasks. Sound engineers rely on expansive instrumentation to aid their audio work. Seeing audio signals is invaluable for algorithm developers. Data visualization can be used creatively for artistic purposes.

Kronos is an emerging programming language and a compiler, designed from ground up to realize audio signal processors from a visual representation. The core technology has been previously presented[1], and the focus of this paper is the visual aspect – designing algorithms in the visual domain.

The rest of this paper is organized as follows: In Section 2, *Advancing Visual Progiramming*, the state of art in visual programming is described, along with novel contributions. Section 3, *Graphics from Signals*, presents a visual domain language within Kronos that is used to build custom data visualization solutions. Several examples of signal visualization are given in Section 4, *Case Studies*, before the *Conclusions* presented in Section 5.

2. ADVANCING VISUAL PROGRAMMING

2.1. A Brief Survey of the Field

Visual programming of musical applications has been examined in depth. A de facto standard in custom signal processors seems to be the commercial Max/MSP by Cycling74 and its open source relative, Pure Data[2]. The success of these systems is indicative of the demand for an easily approachable, visual programming platform for signal processors.

Until recently, these systems have not supported truly comprehensive customization of signal processors, mostly due to the fact that they operate on buffered signals and the associated lack of low-delay recursion. Traditionally, for such customization, a C-language extension to the system would have to be made, rendering the main systems little more than an extensive patching and routing tools for pre-made modules. Such tools are invaluable, and this should be taken as an observation rather than a criticism.

Max/MSP has recently gained the *gen*-addon, which enables visual programming of signal processors on a very detailed level. However, the programming model differs from the main environment, essentially representing a distinct programming language. At the moment, the appeal of *gen* to the community is still hard to gauge, but its inclusion indicates that per-sample processing was desirable for the authors.

Finally, *abstraction* is fairly absent in both Max/MSP and PD. It can be argued that this can be a good thing, as it may lower the learning curve for beginners. However, it also enforces a cap on programmer productivity as her skills increase. The author believes this to be key in advancing visual programming – how to leverage abstraction, make it seem natural in the visual representation, while remaining approachable to non-programmers.

The lack of abstraction in Max/MSP and related platforms also seems be a wider concern. Odot[3] is a project integrating instance-based object oriented techniques into Max/MSP, with potential to improve programming efficiency. However, Odot is not accessible to *gen*, rendering it less useful for aiding low-level signal processing tasks.

Similar limitations and distinctions are present in PWGL[4], the origin of this research. The main system consists of a Lisp-based patching environment, but for audio processing, a separate, less capable domain language called PWGLSynth was implemented for performance reasons.

2.2. Features of *Kronos*

The aim of Kronos is to introduce a visual language that is capable of extremely low level programming and high performance, even when abstract high level constructs are used. Further, these abstractions should be suited to visual representation, which often suffers from apparent one-to-one relationship with physical objects. This subsection briefly describes the core technology. For an extended discussion, the reader is referred to previous publications[1].

Kronos is realized as a client-server architecture. The backend features the compiler and I/O, such as MIDI and audio. The patching frontend is implemented in Microsoft Silverlight, and connects to the backend over OSC[5]. Visual patches are translated into textual Kronos programs and compiled on the fly.

The basic appearance of the Kronos patcher should be familiar to most readers. Processing nodes are inserted on a canvas and connected via patch cords. A special feature is node embedding, allowing the user to dock nodes inside other nodes, where

normally a patch cord connection point would reside. This can greatly reduce visual clutter.

Users are free to type freeform text into nodes. This facilitates the entry of constants, literals and compound expressions. Freeform entry is aided by an interactive search box, which can be used like a searchable menu of built-in modules by beginners. Example nodes are shown in Figure 1.

Figure 1: Example nodes.

2.2.1. Functional programming

For the programming model, Kronos taps into the theory of *functional programming*[6]. A key feature of functional languages is that functions can be used as data. In Kronos, a reverb algorithm could be designed to receive a damping filter from an inlet, subsequently applying it to the internal feedback paths. A more typical example would be algorithmic routing; higher order functions can configure parallel or serial routings based on a I/O configuration using an user-supplied processor.

An example of a higher order function is given in Figure 2, where *Algorithm:Map* receives an anonymous function, *x into Square root of x*, and applies that transformation over a vector connected to the second inlet. The transformation could equally well be and audio process such as a filter.

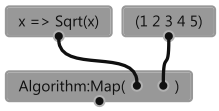

Figure 2: A simple application of functional programming.

2.2.2. Types and Polymorphism

Kronos functions are *generic*. That means that both the functions in the built-in library and those specified by the user can operate on a wide range of signals. It is possible to specify a processor that can accept monophonic or multichannel signals, single or double precision, even real-valued or complex – along with any combination of these.

It should be noted that during execution, Kronos programs are statically typed. This is for performance reasons; it allows for the entire program to run deterministically and requires no dynamic memory allocation. This restriction extends to functions; their output type must be uniquely determined by the input type. This rigidity is what allows the highly abstract, generic language to perform efficiently.

The type system emergently provides a *closure* mechanism. New functions can be constructed on the go, with captured variables or more appropriately, captured *signals*. Such closures are particularily useful when leveraged in conjunction with higher order functions. A large closure is displayed in Figure 6, the final case study of this paper.

2.2.3. Reactivity

Kronos specifies a synchronously updating graph, requiring no manual *bangs* or scheduling aids. Multirate functionality is offered by the means of *clock sources*, functions that cause the patch downstream from them to update at a certain rate. Such sources can be anything from oscillators, audio file players and MIDI inputs to user interface sliders. Computations are only performed when their clock source ticks, for maximum efficiency.

2.2.4. Interoperability

The default mechanism for running Kronos patches is immediate execution. Patches are translated into performant, native x86 machine code upon playback, resulting in efficient utilization of computing power.

However, it is often desirable to be able to integrate the patches into third party code. For this, the backend is also capable of generating C++ code from the user patch. The resulting code can then be included in an audio plugin or a standalone application without introducing any dependencies to the compiler.

The Kronos environment can thus be used to prototype and test an audio processor interactively in the immediate mode, subsequently exporting it to C++ code.

3. GRAPHICS FROM SIGNALS

With the recent research emphasis on the visual user interface, it is natural to focus on the field of signal processing known as data visualization. A data visualizer generates graphics from data. In the case of audio signals, it can be envisioned as a signal processor that accepts audio signals and outputs a graphical representation.

3.1. Signal Routing and Division of Labor

As the Kronos core technology is agnostic to I/O, graphics appear as just another signal to the backend. Each update of an audio signal carries just a single sample of data, but the updates happen very rapidly. A frame of graphics, on the other hand, contains a lot of data, updated at a lower rate.Thus, audio signals are *narrow and fast*, while graphics are *wide and slow*. Generating graphics from audio thus often implies signal rate decimation or buffering.

In modern systems, it makes sense to offload graphics to a dedicated accelerator. The desired output format from the backend is thus a minimum set of parameters and an annotation of how they should be drawn. For example, a graphical polygon should be expressed as a polygon tag, followed by the list of vertices that define it. A graphics subsystem receiving this data is then able to realize the graphics.

A general principle of an audio visualizer is thus a signal processor that inputs an audio signal, outputting a set of vertices and colors.

3.2. The Metadata Protocol

It turns out that the extensible type system in Kronos is also suited for encoding graphics. Drawing annotations can be encoded as user types, containing vertex lists, colors, coordinates and sizes. Being algebraic[7], Kronos types are hierarchical, just like a typical visual tree in a graphics engine.

The visual tree is therefore neatly created by the type inferral process Kronos applies to the patch. This static type defines the

hierarchy of the visual tree, while the instances of the type that stream in from the backend represent graphical frames. Appropriate portions of the incoming stream must thus be routed to the correct nodes of the visual tree.

A metadata protocol is defined for this purpose. Kronos is able to encode its internal types in XML format, where each algebraic junction of the type has a corresponding XML tag. The backend will transmit this XML-encoded type information to the graphics subsystem when the patch is compiled.

The current implementation of the graphics subsystem uses the XML serialization engine in Silverlight to obtain a visual tree decoder directly from the XML description, when all user types used in the Kronos code have a corresponding Silverlight implementation. The decoder is capable of both constructing the visual tree and relaying data to it from the update stream.

As an optimization, invariant parameters are encoded in the metadata and omitted from the per-frame updates.

3.3. A Domain-speficic Language

The graphics language in Kronos is defined by the graphics primitives type set and their associated constructors. The visual tree is defined entirely by the hierarchical type constructed by the patch as the result of type constructors and compositors of various categories.

3.3.1. Atoms

The atoms of which shapes are assembled consist of a *Point* and a *Size* in 2d space, as well as a *Color*.

3.3.2. Shapes

These types define the primitives of which graphics are assembled. *Rectangle* and *Ellipse* are common shapes. For more general tasks, *Triangle-Strip* and *Triangle-Fan* are provided for efficient convex polygons.

3.3.3. Brushes

A brush node can apply a *Fill* or a *Stroke* color to all of its hierarchical children. The effects of these nodes carry over descendants, so that the nearest ancestor takes precedence.

3.3.4. Transforms

These nodes provide transformations in 2d space. *Translate*, *Rotate* and *Scale* are provided. The transformations apply to all their hierarchical children, and can be stacked. Transforms are useful especially when assembling animated visualization.

3.4. Limitations

The fundamental limitation of the graphics subsystem derives from the Kronos type system. During execution, Kronos programs are statically typed. Since the structure of the visual tree is defined by a type, it must also be static during processing. Therefore, nodes can't be added or removed from the visual tree without rebuilding the patch. Likewise, a *Rectangle* cannot dynamically change into an *Ellipse* or similar.

This limitation is not fatal in a data visualization solution. In Section 4, the capabilities of the language in audio signal metering are demonstrated.

4. CASE STUDIES

4.1. Peak Meter

As the first example, the simplest possible peak meter is shown in Figure 3. An audio file player acts as a signal source. To obtain peak levels, two stages of decimation are performed. The first multiplexing stage, *Reactive:Mux*, buffers 16 samples between clock updates.

Subsequently, the maximum value in the buffer is obtained by reducing the buffer with *Max*. To further bring down the signal rate, this lower-rate peak value is again buffered and reduced, resulting in a total decimation factor of 256. While thiscould be done in a single stage, factoring it in two yields – in principle – better low-latency performance due to amortization.

The resulting peak signal is converted into graphics by simply connecting it to the width of a rectangle, resulting in a momentary positive peak meter.

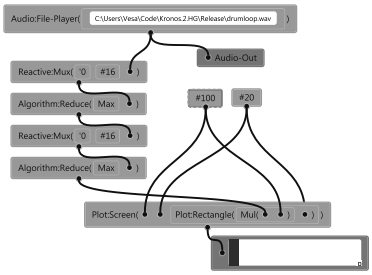

Figure 3: A simple positive peak meter.

4.2. Wave Scanner

As a more advanced example, a scanning oscilloscope is presented. The visualization function is shown in Figure 4.

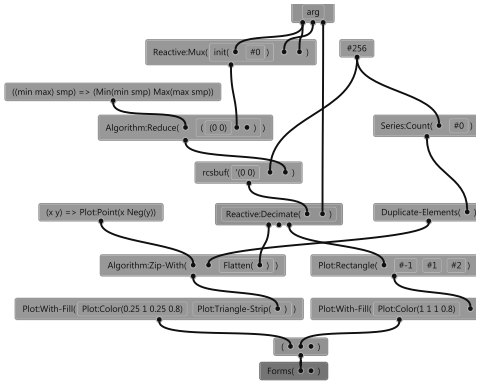

Figure 4: Converting an audio signal into waveform graphics.

The output plot of the function is shown in Figure 5. Initially, the signal is buffered as in 4.1. However, this time reduction is performed by an anonymous function that obtains both the minimum and maximum sample found in the buffer.

These min-max pairs are fed into a ring buffer, which in turn outputs its entire contents along with the current write head position. The update rate is then decimated to reduce computational load. Finally, an evenly spaced, invariant X-coordinate list is generated with *Series:Count*. Each element is duplicated, since both minimum and maximum peaks should have a X-coordinate.

A vector of points is generated from the contents of the ring buffer by *Algorithm:Zip-With*. This function combines the ring buffer content, flattened so that it contains the maximum and minimum peaks in an alternating sequence, with the X-coordinate vector. A zipping function that constructs a *Plot:Point* from the coordinates is employed, resulting in a list of coordinates.

This list is then used to construct a triangle strip which, when plotted, appears as a polygon in the shape of a waveform. Finally, a rectangular cursor is overlaid, placed to indicate the current write position in the ring buffer.

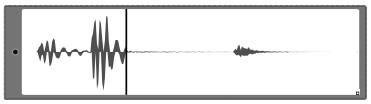

Figure 5: A sample waveform image generated by the visualizer.

4.3. Advanced Visualization

In the final example, a slightly less conventional visualization is presented. While not very useful as such, the example is conceived to demonstrate the power of higher order functions in visualization.

The audio portion of the patch is a resonator on noise, modulated by a slowly changing random signal. This results in a simple wind effect. The wind is visualized graphically by a fractal tree.

At the heart of the fractal is a closure that constructs two translated, rotated, scaled copies of any graphics fed into it, adding a root segment. *Algorithm:Iterate* repeats this closure six times, starting from a simple rectangular segment. As a result, the plot displays a segmented tree, bisecting at each junction.

The rotation and scaling parameters are set by several user interface sliders and captured within the closure. By modulating both rotation parameters with the signal that controls the resonator pitch, the tree is made to bend in time with the artificial wind.

The patch is surprisingly compact, utilizing the power of higher order functions and recursion to generate an extensive data set from humble origins and a limited set of operations.

5. CONCLUSIONS

This paper focused on the visual aspect of Kronos. Although mainly a signal processing language, this paper aims to demonstrate that Kronos is flexible enough to provide for user-designed data visualization.

Specifying data visualizers may be out of reach for beginner programmers, but intermediate and advanced users should find transforming signals into graphics seamless and easy. The Kronos type system provides a natural mechanism for constructing the hierarchical visual tree used as the basis of plotting.

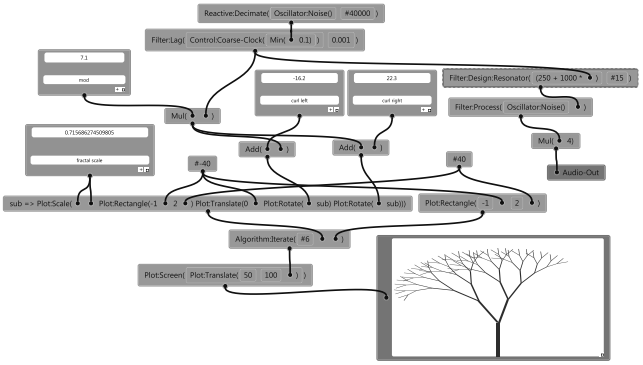

Figure 6: A fractal tree deformed by a control signal shared with a resonator.

The visualizers themselves demonstrate several unique features of the Kronos language. It offers abstraction and programming constructs rarely found in visual languages for signal processors, such as higher order functions and closures[6]. Their application to audio-only processing has been discussed previously[1].

Kronos is to enter open beta in the Spring of 2012, available for Windows and Mac OS X. Meanwhile, videos of the visualization patches can be viewed at http://www.youtube.com/user/vnorilo.

6. REFERENCES

[1] Vesa Norilo, "Introducing Kronos - A Novel Approach to Signal Processing Languages," in *Proceedings of the Linux Audio Conference*, Frank Neumann and Victor Lazzarini, Eds., Maynooth, Ireland, 2011, pp. 9–16, NUIM.

[2] M Puckette, "Pure data: another integrated computer music environment," in *Proceedings of the 1996 International Computer Music Conference*, 1996, pp. 269–272.

[3] Adrian Freed, John MacCallum, and Andy Schmeder, "Dynamic, Instance-based, Object-Oriented Programming (OOP) in Max/MSP using Open Sound Control (OSC) Message Delegation," in *ICMC 2011*, Huddersfield, England, 2011, ICMA.

[4] Mikael Laurson, Mika Kuuskankare, and Vesa Norilo, "An Overview of PWGL, a Visual Programming Environment for Music," *Computer Music Journal*, vol. 33, no. 1, pp. 19–31, 2009.

[5] Matthew Wright, Adrian Freed, and Ali Momeni, "OpenSound Control: State of the Art 2003," in *Proceedings of NIME*, Montreal, 2003, pp. 153–159.

[6] Paul Hudak, "Conception, evolution, and application of functional programming languages," *ACM Computing Surveys*, vol. 21, no. 3, pp. 359–411, 1989.

[7] Konstantin Läufer and Martin Odersky, "Polymorphic type inference and abstract data types," *ACM Trans Program Lang Syst*, vol. 16, no. 5, pp. 1411–1430, 1994.